AUTOMATIC CONCURRENCY GENERATION THROUGH COMMUNICATION DATA SPLITTING BASED ON UML/MARTE MODELS

ABSTRACT

With the increase in the number of processing elements integrated in HW platforms, the development of new solutions helping engineers to design concurrent applications is gaining greater interest. Code refinements required to parallelize sequential algorithms are usually quite complex, and do not guarantee that the decisions taken will provide the expected performance result, requiring a costly and iterative design process. To solve this problem, this paper presents a methodology that enables designers to define and automatically create different concurrent architectures for the system, only modifying different parameters in the UML model. The main idea is to automatically modify the communication between components, adding to the typical configurable options of performing synchronous, asynchronous or buffered calls the idea of enabling parallelization by dividing one service call into several concurrent calls, each one operating with part of the data. Examples of use of this idea with video applications are presented, demonstrating the performance improvement obtained in the final system.

Index Terms— UML/MARTE, CONCURERNCY, CODE SYNTHESYS

1. INTRODUCTION

The evolution of fabrication technologies is enabling the development of increasingly powerful chips containing multiple processors. These chips support different types of architectures, from symmetric structures, to heterogeneous implementations, including CPUs, DSPs or GPUs. As a result, the development of concurrent applications capable of taking advantage of all the computational power of these chips is becoming a critical issue.

However, transforming sequential codes in concurrent applications, while also satisfying the associated real-time constraints is a very complex process. The reason is that this process usually implies two different steps. First, it is necessary to generate the refined concurrent code. After that, designers have to evaluate whether the resulting solution is really capable of obtaining maximum performance for the multi-processor HW. Since adequate management of fork-join solutions, ensuring correct synchronization, is a real challenge on its own, the combination of both steps usually leads to a very costly iterative process.

To deal with this issue, we propose taking advantage of the latter ideas for designing at high abstraction levels. This approach provides an effective way of managing different results while maintaining the global perspective. Then, the development of tools that automatically apply all the details reflected in the high-level model in the final code generation can help designers to face the challenge of overcoming system concurrency.

One of the most common approaches applied for high-level system modelling is the use of UML. However, UML is usually too generic for its application to specific domains, such as embedded systems. To get round this limitation the OMG has proposed a standard called MARTE, oriented to enabling designers to include all the information related to embedded, real-time system designs in the UML models. From this input, some tools are appearing, proposing solutions to apply the information in the model to final implementations through code synthesis. For example, it is possible to define different characteristics in system communications such as synchronism (synchronous/asynchronous) and buffers (fifos) [6]. As a result, designers can modify some channel semantics in the model and automatically obtain wrapper codes that enable direct evaluation of the modelled solution in the target board.

In this way, the paper proposes a solution that goes beyond that initial approach. The basic idea can be easily shown with a simple example. When analyzing images, it is usually possible to divide the image in parts, operate with the different parts separately and then join all the results to continue further processing. Following this idea, we propose a solution where designers can automatically provide information in the UML model. From this model, the infrastructure developed automatically generates wrappers capable of implementing different communication solutions, such as dividing the information sent in a service call in order to launch multiple copies of the service receiving part of the data, and joining the results at the end. Additionally, this approach can be chained when one service calls another service and so on, the final system being a very complex concurrent architecture.

This technique has the advantage that the concurrent architecture is mainly integrated in the automatically generated communication wrappers. As a result, it is mainly hidden from the functional code directly developed by the designer, greatly reducing designer effort.

For this purpose, the paper starts by presenting an introduction to the state of the art in the area, and then it introduces the global methodology; how to integrate all the required information for data splitting into the UML model, and, finally, how the tool automatically creates the wrappers

required in order to generate the resulting concurrency. Finally, some results and conclusions are provided.

2. STATE OF THE ART

The development of tools that enable automatic generation of communication and concurrency infrastructures for component-based systems is a clear need for SW designers who wish to reduce their effort in the implementation of complex systems. To achieve this goal, model-driven design methodologies are commonly adopted to handle the design of large functionalities [5]. The latest design methodologies start from high-level UML models combined with algorithmic codes (e.g. C, C++, Matlab, etc.) of the different system components [16].

In addition, from these UML models, synthesis tools can extract the information necessary to explore different configurations in the model. These configurations can include changing the resource allocation, the application's concurrent behaviour, and the communication semantics.

In this context, UML is a very common way to handle the design of embedded systems. [11] and [12] focuses on the importance of model-driven architectures. [11] pays special attention to the importance of UML models for industrial applications and the effort that they require, and [12] is focused on a particular aspect of MDE concerning model transformations and code generation. The basic model-driven architecture pattern requires the definition of a platform-independent model (PIM) and its automated mapping to one or more platform-specific models (PSMs).

Moreover, different semantic shortcomings have been detected in UML, resulting in the generation of different profiles for specific application scopes. At this point, MARTE [1] represents the OMG standard profile for Modelling and Analysis of Real-Time and Embedded systems. Using it, [14] proposes a methodology for the design of real-time embedded systems which supports UML and the MARTE profile for system modelling.

From the point of view of automatic code synthesis from UML models, [7] focuses on the automatic generation of communication from UML communication models to generate test cases, to run programs and to obtain the execution traces and compare with the diagram activities. There is a proposal of Interface Generation in [8], for Incompatible Intellectual Properties. Its aim is to provide System on Chip designers with a UML-based environment. They model the IP as a UML component with a well-defined interface. However, these approaches are oriented to automatic generation of test cases and interfaces for verification, and do not allow structure systems to be designed at high-level.

In [9], an approach to bridge the gap between UML modelling and SystemC-based verification and synthesis environments is presented. In [10], another approach enabling generation of complete synthesizable HDL code from UML models is presented, while [15] presents a methodology which starts from UML sequence diagrams with MARTE timing constraints and generates VHDL models with checkers. All of these are more focused on diagrams of communication and sequence in order to generate automatic code. In this paper the code is generated from the model elements (i.e.: interfaces, channels, components) captured in UML class diagrams and UML composite structure diagrams.

Several UML-based methodologies also focus on HW/SW communication synthesis. In [17], a semi-automatic solution using Remote Method Invocation (RMI) semantics for generating HW/SW infrastructure from UML models is presented. In [18], a method for synthesizing interfaces for integration of heterogeneous IP (Intellectual property) based on UML models is proposed. The framework supports both interface protocol customization and glue logic generation, thereby maximizing IP integration. Reference [13] describes a technique to synthesize multitasking support and communication infrastructure from UML-ESL description for virtual platform simulation.

Nevertheless, these approaches are limited when designers need to optimize the concurrent architecture of the system. Thus, the proposed approach enables the definition of channel semantics (capacity, pool resources, blocking and non blocking calls, restrictive access to interface functions) to control the behaviour of the application through data splitting, in addition to enabling automatic concurrency generation.

3. AUTOMATIC CODE GENERATION

This paper presents an approach oriented to automatically generating concurrency from UML/MARTE models. It extends the paper presented in [6], where the initial ideas were presented, providing complete solutions for concurrency generation, such as the use of data splitting, and providing more results.

The proposed approach starts from a platform-independent model of the system. In this model, the functionality is structured in components that provide and require services (Figure 1), following a component-based methodology. Additionally, in order to support the mapping to non symmetric systems, to ensure correct access to shared information and to enable communication optimization, the system mapping is performed in two steps: first components are mapped to memory spaces and then these memory spaces are mapped to resources.

Considering a HW platform and a resource allocation, the implemented framework automatically generates the wrapper codes required to execute and interconnect the different services, in order to obtain the implementation binaries. As a result, it is possible to increase the concurrency of the final system in order to optimize the use of the target platform. In order to provide this additional concurrency, the infrastructure takes advantage of the information specified in the communication interfaces.

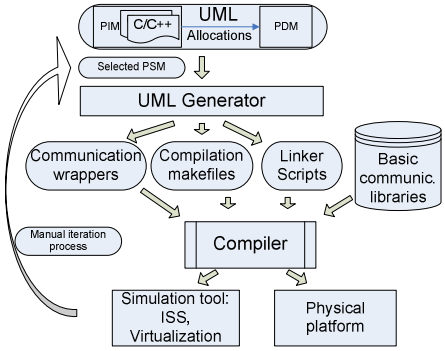

Figure 1: Proposed Synthesis Flow

The resulting communication infrastructure supports the definition of synchronism (synchronous/asynchronous) and buffers (fifos) to implement a pipelined architecture, but it also provides a technique for splitting data transfers in order to enable computing in parallel part of the transferred data-streams (Figure 2). The user can combine all these communication semantics in the UML model, and then the infrastructure generates the final implementation also considering the mapping of the components, using inter-thread, inter-process or inter-node communication services as required.

From this UML/MARTE model, an infrastructure integrated in Eclipse generates all the elements necessary to create the binary files required for simulation or execution in the physical platform. The graphical tool used to create the UML/MARTE model is Papyrus [2]. A code generator has been developed as a set of generation templates written in the standard MTL language [4]. The development has been done through Acceleo [3], a code generation framework fully integrated in Eclipse.

Two kinds of elements are directly generated from the information included in the UML model. First, the C wrappers are generated. These wrappers run the functional code and communicate the interfaces of the components in different memory spaces using the resources of the platform. Synchronization and concurrency management is also included in these wrappers.

Secondly, “makefiles” and linker scripts are generated to enable automatic execution of the compilation processes.

As a result, one executable is obtained for each memory space at the end of the generation process. The executable is compiled for the target HW resource where it is allocated. To achieve this, the generated wrappers contain the entire SW infrastructure required for each memory space to operate and communicate with the other executables.

In this process, the generation of concurrency, takes advantage of the information on component communications described by channels, interfaces, data types and resource allocations in the UML/MARTE model [6].

4. CONCURRENCY GENERATION

Concurrency is generated automatically from the application of the different communication architectures described in Figure 2. From the single executable flow produced by an initial, sequential code, the definition of asynchronous calls and pipelines provokes the generation of additional threads. An asynchronous call requires a new thread, in order to enable the client to continue with its task while the server executes the service. In a similar way, buffered channels enable the implementation of pipelines, where each component has its own execution thread, reading and writing in channel fifos independently of the client operation. At the same time, data splitting enables multiple instances of the same service to be run in parallel, requiring an additional thread per service.

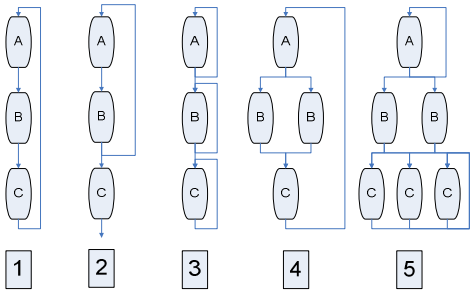

Figure 2: Supported architectures: Sequential (1), Asynchronous (2), Pipeline (3), Data splitting (4), Combined(5)

As long as channel semantics require concurrency generation, resource allocation has the same effect. While the execution of service calls provided by a component running in the same process can be performed by the calling thread, calls to services in other processes or other OSs require a concurrent infrastructure to look for incoming requests and launch the execution of the services needed.

The ad-hoc automatic generation of communication wrapper codes considers all these possibilities in combination, so it is an extremely difficult task. To deal with it, the proposed approach is based on a three-layer method. All communications on both the client and the server side are implemented handling concurrent generation in the highest layer, communication stack management in a second layer, and communication transfers in the lowest layer.

Thus, the first layer is completely dependent on the communication semantics, but independent of the resource allocation. The second layer generates all the resources required to handle data transfers depending on the allocation, but mainly independently of the channel semantics. Finally, the third layer contains the basic communication mechanisms that are only dependent on the platform architecture, but not on allocation or communication semantics.

Among these basic communication functions, the first possibility is that both components are in the same memory partition, the communication is direct. The second

possibility is that the communication is performed between components in different memory spaces, but the memory spaces are allocated in the same OS. Thus, data is passed through OS calls, such as FIFOs. Finally, the allocation of the memory partitions can be in different operating systems. Then, a package transfer solution such as TCP/IP is used.

5. CHANNEL IMPLEMENTATION

The model description is based on components which are connected by channels, providing services. These services are grouped in interfaces. Additionally, for each channel, there is one end which requires an interface and another end which provides the interface.

Thus, the communication mechanism is defined in the UML/MARTE model as the tuple composed of the channel and its supported interfaces. Thus, the semantics of each channel can be easily changed by modifying the semantics of the attributes of a channel and the types of the interface.

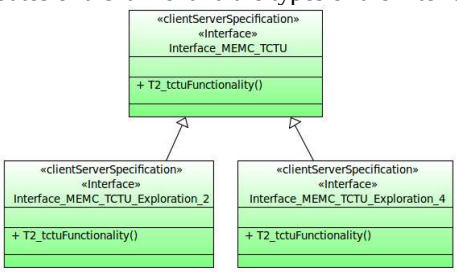

Figure 3: Interfaces inheritance

Specifically, it is possible to modify the types of blockade associated with the function requests communicated through the channel (attributes *blockingFunctionCalls*, *blockingFunctionReturn*) and to consider the capacity of the channel to store service requests (attribute *resMult*). According to these channel attributes, the client application component can be blocked or not until the function is attended, waiting for function returns, modifying the client's concurrent behaviour. On the server side, each request produces a new thread to enter in the component. However, the maximum number of threads in an application component available to attend to requests can be specified in the model (attribute *srPoolSize*). This application component characteristic can model different internal concurrent structures of the application component, producing a significant impact on system performance.

Other features that can be specified for the channels are the priority of the function request being dealt with by the server application of this channel (attribute *priority*) and the maximum time for a function request to be completed (attribute *timeout*). The priority is used by the server scheduler to integrate and sort incoming requests received from different sources. When the server has an available thread in the pool, the scheduler launches the request from the channel with highest priority. The timeout is used to awaken the thread suspended by the blocks described above.

Finally, the interface types can also define the semantics of the services included in them, either sequential, guarded or concurrent [19]. Each interface can only have operations with one of the previous semantics.

The combination of the interface semantic features and the channel semantic features enables the possibility of modelling a wide set of different, communication alternatives. In order to automatically integrate all these semantics in the final binary without modifying the user-provided, functional code, functions are redirected at compilation time. This means that, when a component needs to execute a service from another component, a compiler macro makes it call an alternative function, instead of the service itself. This new function generates a request packet according to the information about interface service and writes it in the channel, generating the associated concurrency. The channel manages the packet and sends it to the provided application component. Then, the provided component receives it according to the information from the request packet and executes it when possible. Finally, the server component sends the response through the channel and the required component updates its execution flow.

5.1 DATA SPLITTING

In order to increase generated concurrency, data packets sent in a service request can be split into parts, enabling several concurrent requests to be called. This data stream splitting is used to provoke the automatic call of multiple concurrent copies of the same function, which involves the creation of multiple threads in the application components.

In the UML/MARTE methodology, interfaces are used for enabling the splitting of data streams since they can be typed. The idea is that, if the interface on the client side has greater data sizes than on the server side, then data streams generated by the client are split considering the data sizes accepted by the server. Thus, the UML/MARTE interfaces enable the definition of the different data types required for specifying the parameters of the interface functions. The UML Data Types are used to define specific data types as arrays and structures.

The data stream splitting is captured in the model by inheritance between interfaces. This modelling mechanism enables new interfaces to be defined that are a generalization of a previous one. These new interfaces (in Figure 3, the interfaces "Interface_MEMC_TCTU_Exploration_2" and "Interface_MEMC_TCTU_Exploration_4") differ from the previous interfaces (in Figure 3, the interfaces "Interface_MEMC_TCTU") in one parameter. Specifically, the difference among these interfaces is the size of the data type that specifies a parameter. In the case in Figure 3, a parameter called "explor" that is typed by different array data types with different size.

Additionally, the inherence of interfaces can be used for joining concurrent independent flows. In order to be executed, an application component should have available

data to be triggered. However, these data come from two different, independent, concurrent flows. In order to specify that a parameter represents an element to be joined, the corresponding join parameters have to be specified in the generalized interfaces; only these parameters have to be specified in generalized interfaces. Then, these parameters are not typed with data type. In this case the code generator knows that the parameter is for joining concurrent flows.

The combination of the previous concurrency specification mechanism provides the designer with a wide palette of modelling resources in order to specify different concurrent structure alternatives in an easy and fast way.

For each component a file is generated. This file contains the execution threads of the component and the functions to manage the channel ends. On the one hand, if the component is a server, the component contains a function to manage the incoming requests for each channel and another to schedule the services to carry out. On the other hand, if the component is a client, the component has a function to manage the responses of the channel. These functions are the functions triggered by the main function.

Finally, wrappers are automatically generated. There are two kinds of wrapper, either provided or required ones. On the one hand, the required wrapper is a file with a function for each interface service function that encapsulates the original call of the function and generates a request structure with all the information about the required service (function identifier, parameters, etc.), and finally sends the structure information through the channel. On the other hand, the provider wrapper file has a function which takes the information from the incoming request to execute the correspondent service.

6. COMMUNICATION LIBRARY INTERFACE

The communication library is the complement of the C code generator. The library is implemented in C code too, and it consists of a set of C files that implement the functions in order to perform the communications and the management of the channels at both ends (required and provided). This library facilitates its reusability and minimizes the amount of generated code lines.

6.1 DATA PARTITION IMPLEMENTATION

As mentioned above, data partitioning is performed by the different parameter data sizes of the interface provided and interface required.

The channel checks the size of the interface service parameters and this produces alternative situations. On the one hand, we have the situation where the data size of the interface's required service is equal to the size of the data in the provided interface service. This situation does not produce extra operations. On the other hand, the size of interfaces services can be different. Firstly, if the required service size is greater, the channel generates a proportional number of request structure children with the same information as the parent, only changing the data to be partitioned. Then requests are performed as described in previous sections. When the requests are completed, the channel modifies the data of a parent one and frees the memory of the partial requests. Secondly, if provided size is greater, the channel waits for all the requests to arrive. When they are all available, the channel generates a parent request with the information and merges the data. A thread service is then executed to deal with the parent request and, when completed, the channel updates the request parameters of child requests.

6.2 JOINING CONCURRENT FLOWS

After splitting data and performing concurrent calls, it is usually necessary to merge these flows to continue the execution of the system. This mechanism enables the synchronization between two concurrent flows of execution when they require the same interface and this interface uses parameters from these flows.

The generated wrappers create request structures with the parameters of the parent interface and the parameters of availability are set to null. Then the wrapper generates a function to synchronize both flows until all parameters are available. This synchronization is performed with semaphores and all flows are blocked except one which has access to the parent service interface: When the service finishes, the other requests are awaken.

7 USE CASES

An MPEG-4 encoder implementation is used in this paper to asses the proposed methodology. It enables different system configurations to be established by modifying the channel semantics of the model. The MPEG-4 encoder is an industry-standard, consisting of a motion-estimation and compensation (MEMC) phase followed by transformation (TCTU) and entropy coding (EC) phases. Finally, the data is packed (BP) (Figure 4). From an initial sequential implementation, channel semantics enable the definition of different parallel regions operating with split data. Automatically generated code controls the concurrency, data management and synchronization required to interconnect all the components according to Figure 4.

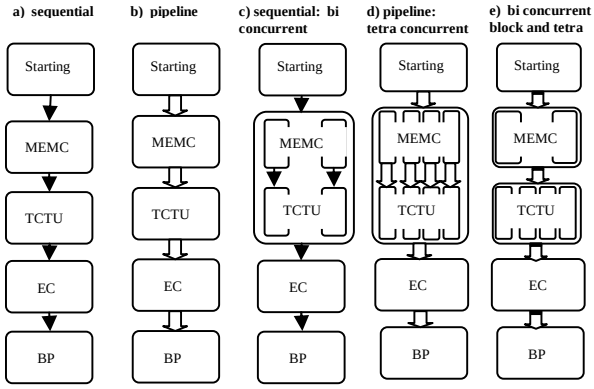

Figure 4: Explored MPEG-4 Configurations

In case a), clients always wait for server completion to continue. In case b), buffered calls are used to implement a pipelined architecture. The cases c), d) and e) use data partitioning. In case c), the interface definition enables the execution to use two threads in order to compute a frame in MEMC and TCTU, blocking components until finishing the frame encoding process. Case d) uses data partition, generating four threads with pipeline. Finally, case e) uses data partition for two threads in MEMC and four threads in TCTU without blocking.

The results obtained from applying the different configurations in the MPEG-4 encoder were evaluated in the OMAP-4 (PandaBoard) HW platform.

CASE	EXECUTION TIME
a) Sequential	21.05 seconds
b) Pipeline	15.70 seconds
c) Sequential bi concurrent	13.01 seconds
d) Pipeline tetra concurrent	11.11 seconds
e) Bi and tetra concurrent	12.89 seconds

Table 1: MPEG4 Simulation on OMAP-4

In order to test the merging mechanism, a stereoscopic vision use case was applied. This system takes two images that are initially rectified to enable their later comparison in order to extract the position of the different objects of the image with respect to the observer. Both images are initially processing in parallel, so two image processing flows are defined for that purpose, one for the right image and a second one for the left image. Each image is pre-processed, checking whether the image has enough quality. Then, a classical two-frame stereo matching algorithm starts.

The Stereovision use case was executed in different platforms: on OMAP-3 (BeagleBoard) using local communications, on a TCP-IP system between OMAP-3 and OMAP-4 and on a SPEAr-600 (double-core architecture with different Operating Systems) using TCP-IP.

PLATFORM	EXECUTION TIME
Beagle	315.960 seconds
Beagle-Panda	261.161 seconds
SPEAr-600	265.878 seconds

Table 2: Stereovision Simulation

9. CONCLUSIONS AND FUTURE WORK

The paper presents a methodology that enables the application of different concurrency architectures in a system described in UML/MARTE, modifying communication between components. From the UML/MARTE model, some code files are generated in order to compile them with the functional code of the application. The whole C code is compiled with generated "makefiles" and the final binary executables are created for the target platform.

In this context, data splitting is used to increase system concurrency enabling service calls to be divided in a set of smaller concurrent operations.

This methodology can be used to optimize the resources used in embedded systems, since it enables different concurrent architectures to be explored without requiring designer effort, due to the implemented code generation process.

10. REFERENCES

[1] OMG: "UML Profile for MARTE", www.omgmarte.org, 2013.

[2] http://www.papyrusuml.org/

[3] Website. www.acceleo.org. Nov., 2010.

[4] OMG. MOF Model To Text Language. Jan., 2008.

[5] D. C. Schmidt, "Model-driven Engineering" IEEE Computer, vol. 39 no. 2, pp. 25-31, 2006.

[6] Ommited for Blind review

[7] P. Samuel, R. Mall, P. Kanth: "Automatic Test Case Generation From UML Communication Diagrams". Information and Software Technology, February 2007.

[8] F. Boutekkouk, Z. Tolba, M. Okab: "Automatic Interface Generation between Incompatible Intellectual Properties(IPs) from UML Models". Advances in Computing and Communications. Commmunications in Computer and Information Science Volume 191, 2011, pp 40-47.

[9] F. Mischkalla, D. He, W. Mueller: "Closing the gap between UML-based modeling, simulation and synthesis of combined HW/SW systems". DATE, 2010.

[10] T. Schattkowsky, J.H. Hausmann, G. Engels: "Using UML activities for system-on-chip design and synthesis". MoDELS 2006.

[11] W. Mueller, Y. Vanderperren: "UML and model driven development for SoC design". CODES+ISSS'06.

[12] J. Dekeyser, P. Boulet, P. Marquet, S. Meftali: "Model driven engineering for Soc co-design". IEEE-NEWCAS, June 2005.

[13] T.Cardoso, E. Barros, B.Prado, A. Aziz: "Communication software synthesis from UML-ESL models". Symposium on Integrated Circuits and System Design. SBCCI Brasilia 2012.

[14] F. Herrera, P. Peñil, E. Villar, F. Ferrero and R. Valencia "An Embedded System Modelling Methodology for Design Space Exploration". JCE 2012.

[15] E. Ebeid, D. Quaglia, F. Fummi: "Generation of VHDL Code from UML/MARTE Sequence Diagrams for Verification and Synthesis". Digital System Design (DSD) 2012.

[16] Y. Vanderperren, W. Mueller, and W. Dehaene, "UML for electronic systems design: a comprehensive overview," Design Automation for Embedded Systems, vol. 12, no. 4, 2008.

[17] J. Barba, F. Rincón, F. Moya, J.D. Dondo J.C. López. "A comprehensive integration infrastructure for embedded system design", Microprocessors and Microsystems, 2012.

[18] S. Zhenxin, W. Weng-Fai. "A UML-based approach for heterogeneous IP integration". ASP-DAC, 2009.

[19] http://www.omg.org/spec/UML/2.4.1/